Memory Management in Symunix II: A Design for Large-Scale Shared Memory Multiprocessors

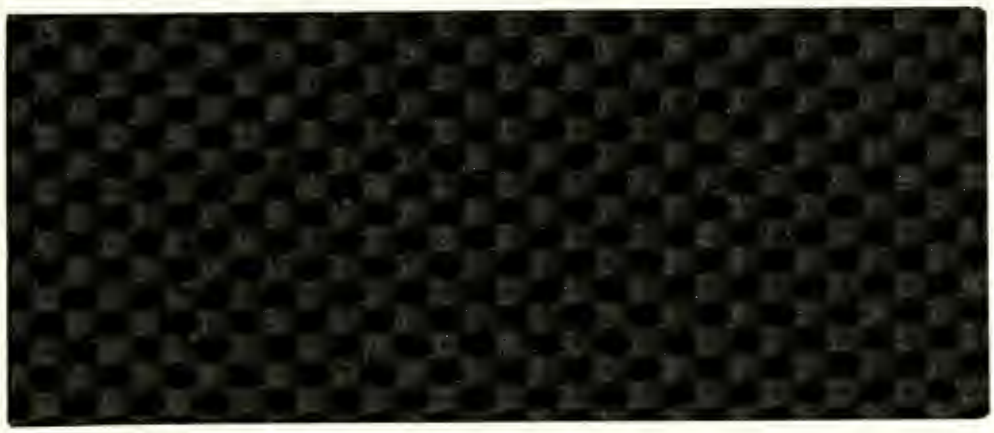
by

Jan Edler, Jim Lipkis, Edith Schonberg

Ultracomputer Note #135
April, 1988

Ultracomputer Research Laboratory

New York University
Courant Institute of Mathematical Sciences
Division of Computer Science
251 Mercer Street, New York, NY 10012



Memory Management in Symunix II:
A Design for Large-Scale Shared Memory Multiprocessors

by

Jan Edler, Jim Lipkis, Edith Schonberg

Ultracomputer Note #135
April, 1988

This work was supported in part by the Applied Mathematical Sciences Program of the US Department of Energy under contract DE-AC02-76ER03077 and in part by contract number W-7405-ENG-48.

ABSTRACT

While various vendors and independent research groups have adapted UNIX and other operating systems for multiprocessor architectures, relatively little work has been done in anticipation of the software requirements of very large-scale shared memory machines containing thousands of processors. Programming environments for these machines must exploit multi-level memory and cache hierarchies so as to obtain the maximum performance available from the hardware architecture. Several years of experience at New York University with parallel systems and programming environments have led to a memory management design emphasizing flexible control over arbitrarily complex patterns of memory sharing and cacheing. This design integrates user memory management, virtual memory management (swapping/paging), program debugging and monitoring, the file system and buffer cache, and checkpoint/restart in a synergistic manner that extends the existing strengths of UNIX. It is being implemented for two research multiprocessor prototypes, the NYU Ultracomputer and IBM RP3.

1. Introduction

Symunix II is a 4.3 BSD UNIX[1] compatible operating system being designed for highly parallel MIMD shared memory architectures, specifically the NYU Ultracomputer [Got87] and the IBM RP3 [PBG85]. Like Symunix I [EGL86], Symunix II is fully symmetric; i.e., no processor plays a distinguished role. Since scaling to a large number of processors (up to several thousands) is a central objective, serial bottlenecks are avoided wherever possible by using non-blocking parallel algorithms and data structures both of which are based on fetch-and-add [GLR83]. A key difference between Symunix I and II is in the area of memory management. Symunix I remains close to the V7 model; the Symunix II memory management design, which is the topic of this paper, incorporates a more flexible virtual memory model and an integrated set of novel features to support large scale parallel computing.

By now, many UNIX-based systems have been enhanced or redesigned for multiprocessors, and support a variety of parallel processing models [BO87, HK86, TR87]. Popular extensions include lightweight processes in the kernel as the unit of scheduling, and shared memory capabilities. Lightweight processes present a parallel programming model in which all resources are shared, including the entire address space [HK86, TR87].

Within the framework of a more standard UNIX process model, in which each process has private resources and its own address space, other explicit mechanisms for sharing have been designed. These include sharing via inheritance (over **fork** system calls) as in SunOS [GMS87] and Mach [TR87]; sharing global memory regions as in System V (where the name of the region is a user-chosen number); sharing memory by using the file system

[1] UNIX is a registered trademark of AT&T.

name space to name shared regions, and file mapping to define the regions, as in SunOS and Dynix [BO87]; and by defining an interface for shared memory object management outside the kernel, as is provided by the external pager interface of Mach. The latter three mechanisms facilitate sharing among processes not related by **fork** alone.

In Symunix II, the unit of scheduling is a full-featured UNIX process. This model is preferred over the lightweight process model both because of the expense of scheduling lightweight processes in the kernel and because of the desirability of per-process private resources (elaborated in Section 2.1). Parallel applications often require application-specific or language-specific task scheduling, and general-purpose, low-overhead tasking packages can be provided in user mode.[2] Within this process model, parallelism is created with **spawn**, a parallel generalization of **fork**. (Parallel programming may be characterized as the one area where **fork** is most often *not* soon followed by **exec**!) We refer to the set of processes related by **spawn** but not **exec** as a *family*; thus members of a family are those that execute the same object file, and a family intuitively is a parallel program. Memory sharing may be provided by inheritance over **spawn**. But sharing via inheritance alone is not sufficient for parallel programming. Therefore, shared memory is also provided through file mapping and the file system name space.

While these shared memory mechanisms are similar to those used in other systems [TR87, BO87, GMS87], the Symunix II memory management design is novel in a number of significant respects, in particular, in the integration of features. The file system serves as backing store for the entire process image (as in Sprite [OCD88]); file mapping is the *only* means by which a process may obtain memory. The apparent performance penalties are mitigated through use of several devices and optimizations, while library packages provide a compatible easy to use interface.

Like other recent re-designers of the UNIX memory management subsystem, we are striving for enhanced performance through modern memory management techniques; clean interfaces between the memory management component and the other levels of the kernel; and, particularly, portability across architectures. Our initial implementation is targeted to three machines with highly dissimilar memory mapping designs: the Ultracomputer prototype known as Ultra II, based on MC68010/MC68451 processors (variable-length segments); the RP3 (fixed-size pages, mapping using segment and page tables); and the IBM RT PC, our token uniprocessor (fixed-size pages, mapping using an inverted page table).

The remainder of the paper is organized as follows. Section 2 outlines the parallel processing requirements that most influenced the design. Section 3 describes the user's memory model and the system interface. Some aspects of the internal design, including performance issues, are presented in Section 4. The concluding Section 5 gives a status report.

---

[2]See [ELS88] for further discussion of these process model issues.

2. Memory Management for Parallel Processing

While many UNIX environments are dominated by highly interactive, short running, non-numeric types of programs, the parallel processing world is dominated by large, long-running, scientific or other compute-intensive applications. Often a single job, consisting of many subtasks or processes, will want to take over an entire computer system for a large chunk of time, with little interference from the operating system. This perspective has sometimes resulted in the viewpoint that there should be minimal operating system services provided on parallel machines — perhaps little more than a batch scheduler.

Our viewpoint is that shared memory parallel machines should have general purpose multi-programming operating systems, and *general* facilities that take into account the special needs of parallel computing. In the area of memory management, there are a number of issues beyond the mere presence of shared memory that must be addressed. We detail them in the following.

2.1. Shared and private memory

Our parallel program model does not preclude hardware-isolated (i.e., separately mapped) private memory. Besides the fact that it is necessary for UNIX compatibility, there are several reasons why private memory may be useful:

- Private memory provides protection from wild stores in which non-shared variables might be clobbered asynchronously by other processes, causing obscure bugs in supposedly safe sections of a parallel program.
- For certain processor architectures, absolute addressing often results in more efficient code. In programs with a single level of parallelism, private variables may be assigned identical virtual addresses that are individually mapped to private memory regions, thus allowing absolute addressing modes.
- Absolute addressing lessens the need for reserved base registers, thus again improving the efficiency of generated code for user programs. This has arisen in the development of microtasking environments.
- Standard UNIX compilers and libraries assume absolute addressing for statically allocated variables. Although a secondary issue, such software is less portable when private memory is unavailable.

The advantages of absolute addressing stem from the fact that the "registers" of the memory management hardware can be used in effect as extra pointer registers by the executing program.

Sharing via selective inheritance over **fork** and **spawn** is a particularly useful mechanism for parallel processing, as it is a simple and clean way for parallel processes executing the same program text to have both shared and private data. Why are other sharing mechanisms needed?

- The inheritance mechanism does not allow shared memory to be dynamically allocated by a process of an existing family, after a spawn has already been executed.
- More sophisticated parallel applications can be larger than a single family, and memory sharing may be desirable between more remotely related processes. For example, a facility for inter-family shared memory allows for IPC implementations entirely in user mode.

Dynamic sharing, and a scheme for naming memory regions, are necessary for addressing these requirements. We have chosen to use the file system for this purpose because it is a rich, structured naming scheme that allows nested scoping of names, provides protection, *and* because it builds on the existing naming conventions for objects in UNIX.

2.2. Flexible low-level interface

Large-scale shared memory MIMD machines will have large virtual and physical address spaces, one or more levels of cacheing, and possibly a hierarchy of memory.[3] On such machines, each region of a process's virtual memory has several attributes, including read/write/execute protection, shared/private scoping, cacheable/non-cacheable access, and local/global physical memory. For dealing effectively with different kinds of memory, the standard UNIX model of fixed text, data, and stack segments is entirely inadequate, being both too high level and too rigid. Because of the wide number of varying requirements of parallel applications, and the many ways a large virtual address space may be split into different flavors of memory, it is too restrictive for the kernel to assign fixed virtual addresses. It is also undesirable for the kernel to attribute meanings to various memory regions. For example, a parallel programming environment may not have use for the standard type of stack that is provided for sequential C, but rather a cactus stack, or multiple stacks allocated within a general purpose heap.

For these reasons, the Symunix II kernel memory management interface is both low level and flexible, and allows for user specification of virtual and physical attributes. The attributes are independent and orthogonal: it should not be assumed, for example, that shared memory is always global (a process mailbox may be shared and local), nor that private memory is always local (in the presence of processor context switching, this may be undesirable). Virtual address space management is the responsibility of applications, libraries, and language environments. This does not necessarily burden the programmer — standard libraries are provided — but rather allows for a diversity of environments.

---

[3]The RP3 has shared local memory, as well as shared global memory, accessed through an interconnection network. The local memory is accessed sequentially, while the global memory is interleaved. Local memory on the Ultracomputer is not sharable. Both the RP3 and Ultracomputer have processor caches.

2.3. User control over cacheability

It cannot be expected that large-scale shared memory MIMD machines with processor caches can enforce cache coherence in hardware. Currently available hardware cache coherence technology, based on inter-cache communication over a bus, does not scale up to highly parallel configurations which are interconnected via multi-stage networks. Both the Ultracomputer and the RP3 allow the operating system to suppress cacheing for certain areas of memory; the cacheability option is specified as a component of the virtual-to-physical memory mapping.

In theory, cache coherence for user programs might be obtained in several ways:

a) Enforce a policy in the operating system whereby all shared memory that is not read-only is non-cacheable.

b) Implement software cache coherence in the operating system. The kernel would trap stores to shared cacheable memory, and immediately notify other processors to invalidate their caches.

c) Do not enforce cache consistency in the kernel, but provide for user mode control over cacheability.

Option a) is not adequate, because there are a significant number of parallel applications for which the ability to cache read-write shared memory is important for performance. A typical example is a parallel program in which each process computes values within a section of a grid represented as a multi-dimensional array. Such applications exhibit great locality of reference, with occasional non-local references to neighboring grid sections. Both for ease of programming and compilation, and to allow for dynamic adjustment of the grid and temporary access of neighboring grid sections, the grid array should be allocated in shared memory. Moreover, because of the local nature of the computation, efficiency suffers greatly if shared writable memory is not cacheable. Chaotic relaxation [Ba78], in which consistency can be eased over several iterations, allows for even more cacheing.

Option b) is very inefficient, because the operating system must intervene on each store to a shared variable and because some stores to shared memory may not require invalidation of other processors' caches anyway, as the grid example illustrates. Furthermore, this is a serial solution which does not scale well. Ultimately, only the application programmer, or optimizing compiler, knows when cacheing is safe and when the caches must be invalidated or flushed. There would be little point in using caches at all if coherence must be simulated in software.

In Symunix II, cacheability is visible in the user interface. A region of virtual memory may be flagged as cacheable or non-cacheable. The attribute may be changed at will, and the user can invalidate and/or flush the processor cache as needed. Finer control, in both space and time, over the cacheability distinction can be obtained using memory region *aliasing*: that is, two different virtual address ranges, one cacheable and one non-cacheable, can

be mapped to the same region of physical memory. When a memory location is cacheable, the cacheable address alias is used; otherwise the non-cacheable address is used. The overhead of switching from cacheable access of a variable to non-cacheable access then involves at most a cache flush and/or invalidation.

An important application of cacheability aliasing occurs in a run-time environment for a block structured language with stack frames. If all physical memory locations have two virtual addresses in the same process, one cacheable and one non-cacheable, then adjacent variables (in the same stack frame or physical page) need not have the same cacheability attribute.[4] The compiler may generate cacheable/non-cacheable references based on what it knows about the status of the variable. Without aliasing, it is more difficult to sub-partition a stack frame according to cacheability properties.[5]

2.4. Process image considerations

The process image includes the actual text and data of a process and also certain data that store execution state information (e.g. registers). The ability to access and examine a process image, both during execution and after termination, is important to support such capabilities as interactive and post-mortem debugging and program monitoring. The need for debugging and monitoring tools is is particularly acute in the more complex parallel programming environment. Furthermore, typical applications are extremely compute intensive, and may run for days or weeks. Being able to conveniently generate restartable checkpoints of process images is thus an important goal.[6]

The standard UNIX **core** file does not adequately represent the post-mortem process image of parallel programs. Since a parallel program is a *set* of processes that share resources, the post-mortem image must reflect the entire set of processes. If all processes in a parallel program are executing in the same current directory and exit abnormally at roughly the same time, they will overwrite each others' **core** file. At best only one of the processes will be represented; at worst, the **core** file will be garbled.

These and similar considerations have led to the following goals for the Symunix II process image representation:

- *Uniformity Of Representation.* The same representation of a process should be used by all components that need to access the process image, including such utilities as **ps**, debuggers, checkpointers, and any other status displays or tools that may be devised in the future. (The loader **a.out** format could also be merged with the uniform process

[4]However, variables with different cacheability must be isolated in separate cache lines.

[5]On the RP3, because the cache provides a partial-invalidate feature, there are actually three classes of cacheability rather than just two.

[6]The only UNIX-based system that we know of with checkpoint/restart is the CRI UNICOS system [HK86].

image representation; this is a potential topic for future work.)

- *Independence From Kernel Storage Structures.* Reliance on /dev/kmem as a panacea for accessing process state is not advocated. In fact, we should have freedom to restructure, eliminate, or add kernel tables with minimal impact on the kernel interface and user mode programs. For example, elimination of the *u*-block should be possible without affecting utilities and tools.
- *Representation Of Parallel Programs.* Since a parallel program consists of multiple processes, it must be possible to reconstruct the relationships among the processes, as well as the actual data of individual processes, from information in a **core** dump or checkpoint.
- *Consistent Representation.* The issue of consistency pertains to the accurate representation of parallel programs with shared memory objects. A representation is *consistent* if it stores a valid state of computation reached during program execution. In particular, if several processes sharing memory core dump or are checkpointed by a signal, the combined saved images should represent a valid computation state. Potential problems arise because signals arrive and are handled asynchronously. If multiple copies of a shared object are made by processes receiving a signal, then copies made at different times are potentially inconsistent; even if there is only one copy of each shared object dumped, care must be taken to avoid inconsistencies between shared and non-shared objects at the time of signal handling.

  Another type of potential inconsistency in shared objects arises when sharing files across a network. If two processes on separate nodes map an address region to the same file (see below), then write operations to the individual buffer areas can result in file inconsistencies. We consider this to be a file system issue: if the network file system supports consistent read/write sharing, then that mechanism can be used to maintain consistency for shared memory segments.
- *Security.* If process images are restartable, care must be taken to prevent security holes while executing **setuid** programs.

The work described in [Kil84] demonstrates that the file system interface provides an easy and general purpose way to access the process image for debugging. We carry this idea further by actually maintaining the entire process image as a set of ordinary files. In Symunix II, the file system serves to unify the treatment of all memory regions in an address space. There is in fact a single representation of a process image for the diverse purposes of core dumping, process monitoring, swapping/paging, and checkpoint/restart. A more detailed presentation of this design is the topic of Section 3.

3. Symunix II Memory Management

In the following, we return to all of the issues discussed in the previous sections, but in a more concrete fashion. The model of memory supported by the Symunix II kernel, as well as the set of system calls provided to manipulate memory are presented, and we contrast this design with related work.

3.1. Memory model

3.1.1. Logical and physical segments

The virtual address space of a process consists of a variable number of *logical segments*, each a contiguous range of virtual addresses. The address space can be sparse; the only restrictions on placement of logical segments within the address space are that they be disjoint and that proper alignment and size restrictions for the underlying architecture be obeyed.

A logical segment is mapped into a physical memory object called a *physical segment.* If more than one logical segment is mapped into the same physical segment, the physical segment is *shared.*

Each logical segment defines a set of *virtual attributes*, which may differ among logical segments sharing the same physical segment. These include the following:

- Protection: a process can have read, write, and/or execute permission on a logical segment.
- Cacheability: a process may specify whether a logical segment is cacheable or not.
- Action on **spawn/fork**: when a process spawns or forks subprocesses, a logical segment can be (1) dropped from, (2) copied into[7], (3) inherited into (shared with), or (4) reallocated (without copying) in, the address spaces of the subprocesses.
- Action on **exec**: a logical segment can be preserved or dropped from the process image during an **exec** system call. This feature can be used to transmit environment variables in user mode, for example.

It is possible to change virtual attributes of a logical segment, or of a portion of a logical segment. (If attributes of a portion of the logical segment are changed, the segment is split into smaller segments.)

Physical segments can also have attributes. Unlike logical attributes, these *physical attributes* are specified only when the segment is created, cannot be changed subsequently, and may be machine dependent. An example on the RP3 is the physical memory type, which may be *sequential* (local) or *interleaved* (global). A second attribute, *strict* vs. *non-strict*, affects the interpretation of the physical memory specification. A strict allocation

[7]The copy-on-write optimization is employed (see below).

request will fail if the kernel is unable to obtain enough memory of the requested physical type; a non-strict request is advisory in that memory of another kind may be substituted. A non-strict request also permits eventual implementation of sophisticated mechanisms like demand paging of local pages from a larger global memory space.

3.1.2. Image files

Each physical memory segment is described by an offset and a size within a corresponding *image file*. The image file, which is an ordinary file in the UNIX file system, stores the contents of the physical segment. Portions of physical segments are allocated in memory during process execution; thus main memory serves to cache image files. Memory management is thereby integrated with file I/O: every frame of main memory is, at any point in time, either (1) unused, or (2) being used to cache a block of an image file (i.e., mapped into one or more logical segments), and/or (3) being used to cache a block of a file being read or written, replacing the function of the kernel buffer pool of traditional UNIX.

Image files provide a unified framework for a set of distinct features for both parallel programming and general system performance. More specifically, mapped image files provide:

- Backing store for swapping and/or paging.
- A means for loosely related processes to share memory via the file system name space.
- A limited memory-mapped I/O facility for improving random file access.[8]
- A unified process image, consisting of a set of files storing both shared and private data. Image files enable access to the text and data segments of a process during execution, for debuggers and status display programs, and provide a permanent snapshot of the process state, for post-mortem debugging or checkpoint/restart.

There are two mechanisms provided for establishing sharing, as mentioned in Section 1.

(1) A logical segment can have its action-on-spawn attribute set to *inherit*. On spawn, the child processes will share the physical segment/image file into which the original logical segment is mapped, at the same virtual address.

(2) Two or more processes can map into the same image file via the new **mapin** system call. It is only necessary for the cooperating processes to have the appropriate permissions to access the file. Unlike sharing via inheritance, this mechanism provides sharing among processes not directly related by spawn, and it is possible for logical segments sharing the same physical segment to span different virtual ranges.

---

[8]In a swapping implementation, the size of a mapped file will be limited by the size of physical memory. In any case, the **read** system call is still needed for initializing a logical segment from a file, so read cannot be implemented purely on top of file mapping.

Physical memory segments in Symunix II are created only through file mapping. The **fork**, **spawn**, and **exec** system calls use an internal kernel version of **mapin** for this purpose. If a special file, representing a device, is mapped in, then no physical memory is allocated; instead, a logical segment is created and mapped as directed by the respective device driver. Directories may not be mapped in.

The memory management system interface, listed below, specifies basic operations on logical segments. The first three calls modify process address space. The last two calls obtain information about the address space composition.

Our use of image files is similar in some respects to the specified but unimplemented **mmap** system call package from 4.2 BSD, which also maps files into the process address space. Variations on this package have been implemented in SunOS and Dynix, among others. Dynix, however, has a more static view of the partitioning of memory into private, shared, and stack areas than Symunix II. In the **mmap** model, file mapping is used only

mapin(*vaddr, size, imagefd, imageoff, flags*)	Create a physical segment from the image file accessed through *imagefd* at file offset *imageoff*; map it into the virtual segment beginning at *vaddr*. Virtual and physical attributes are given by *flags*. (To initialize from another file, use **read**.)
mapout(*vaddr, size*)	Unmap the virtual address range beginning at *vaddr*. (*Vaddr* need not begin a previously allocated logical segment.)
mapcntl(*vaddr, size, flags*)	Change the virtual attributes of the address range beginning at *vaddr* according to *flags*.
pinfo(*pid, parea*)	Obtain basic (fixed size) information about the process image of process *pid* returned in location *parea*.
pmemaddr(*pid, parea, vaddr*)	Obtain information about the logical segment that includes *vaddr* for process *pid*.
pmemvect(*pid, parea, psize*)	Obtain a vector of information, of *psize* bytes, about all logical segments in the address space of process *pid*.

Table 1.

when memory sharing is intended. Otherwise, memory regions are allocated from *anonymous* storage, which is backed up by dedicated swap areas on disk. In Symunix II, image files serve as backing store for the entire address space; no anonymous swap area is used.

Furthermore, when an image file is mapped to several read/write logical segments in Symunix II, it is fully shared by all of them; writes by each are visible to all. If it is desired that changes remain private, then it is necessary that the processes create private memory segments (i.e., mapin separate image files).[9] We do not provide an analog to the MAP_PRIVATE/MAP_SHARED option of **mmap** as in SunOS, through which some of the sharing processes may request that their writes be propagated to private copies in anonymous storage, while other processes sharing the same pages request that their writes be reflected in the original, shared copy. We also do not provide an **msync** system call to flush modified pages in memory to disk, as in SunOS, since the existing **fsync** primitive satisfies this requirement at the kernel interface level.

3.1.3. User mode address space management

The Symunix II kernel does not interpret particular virtual memory ranges in any special way. Virtual address space management is entirely the responsibility of the user mode layer. This decision was made both to avoid duplication of function and because allocated memory may be shared in complex patterns among many processes. It is undesirable to establish policies in the kernel for coordinating address space layouts among sets of processes.

Thus, while the kernel creates mappings between process address spaces and image files (and thus physical memory), it is left to user programs, compilers, linkage editors, or libraries to establish sharing of memory. In particular, a C library interface, extending **malloc** and **free**, is provided for dynamically allocating private and shared memory that:

- hides explicit references to image files and other lower level system interface details from the user;
- manages the address space of a process or, more importantly, of a family of processes, so that shared memory regions can be allocated at identical virtual addresses within the processes of the family;
- implements sharing, allowing a user to issue a single request to allocate or free a region of shared memory, even after spawns;
- allows for setting up virtual aliases of regions, for specifying virtual and physical attributes, and for mapping regions into files.

[9]Nonetheless, copy-on-write may be used among the private segments, subject to alignment requirements of the underlying hardware, when they are initialized from a common file through **exec** or **read** system calls.

The mechanics of establishing sharing are hidden through a special SIGSEGV handler, which detects when it is necessary to mapin a shared region for the current process. (Dynix uses a similar technique.) Furthermore, since "automatic" stack growth must be handled in user mode, the same package, again using the SIGSEGV handler, allocates memory for stack growth according to the stack discipline in use.

3.2. Management of the process image

During execution, the text and data of the process image are contained in a set of image files, which can be accessed subject to the file permissions. According to local installation policy, image files may be kept in a special directory hierarchy or file system. *Reference directories*, described below, may be used for managing administrative conventions in user mode.

3.2.1. Reference directories

We introduce the notion of *reference directories* largely for performance reasons, but they play a role in the memory management interface as well. Image files are created implicitly by the kernel on **spawn**, **fork**, and **exec**; we need to be able to specify a "current" image file directory, analogous to the traditional current working directory, where these image files will be stored.

Path search, i.e., finding an inode from a file name, is traditionally an expensive process in UNIX, depending on the depth of the directory hierarchy that must be traversed.[10] This search is eliminated for file references relative to the current working directory by keeping the inode pointer for that directory in the *u*-block.

Reference directories extend this feature. Any frequently accessed directory may be designated as a reference directory and bound to a *refname*; its inode is then saved to avoid further path searches. *Refnames* are prefixed with the character '@'. The *refname* **@image** is specially reserved by the kernel to designate the repository for the current process' image files, in particular, for those created implicitly by the kernel. Reference directories are defined and removed via the functions:

mkrefdir(*path, refname*)
fmkrefdir(*fd, refname*)
rmrefdir(*refname*).

Users can call **mkrefdir** to change the **@image** directory, or to define other frequently referenced directories (e.g., directories in $PATH). Files in the reference directory may be accessed with the name *@refname/filename*.

---

[10]In 4.3 BSD, the path search overhead is reduced through use of a name translation cache.

3.2.2. Core Files

An important question is the dispensation of the image files upon process termination. When a file is created solely to act as an image file, it is typically of interest only as long as the processes mapping the file are still running. To avoid cluttering up the file system, such a file should disappear when the last process that maps it terminates. However, there are sometimes good reasons not to delete image files upon process termination. In the event of abnormal termination, image files comprise the process data needed for post-mortem analysis and possible restart.

To allow for user control over the fate of image files, Symunix II defines additional flag options on the **open** and fcntl system calls. These options specify action on final close operation: always delete the file on the last close operation; delete the file *unless* a process that has opened it terminates abnormally; and delete the file if a process that has opened it terminates abnormally. (This feature implies that file names of opened files are remembered by the system.)

When a core dumping signal is received, the kernel creates a file called **core.***pid* that stores the process state information. (As usual, *pid* is the process identifier of the current process.) The information stored should be sufficient for the process to be restarted. The process image text and data are already available in the image files for the process. Moreover, upon restart, shared image files will be consistent because they store the most recent updates to shared variables at the time the last process sharing the files terminated.

3.3. Checkpoint/restart

A checkpoint/restart facility is planned in Symunix II through a new SIGCHKPT signal and a **restart** system call. A core dump corresponds to a *terminal checkpoint*; other checkpoints (triggered by SIGCHKPT) require copying the relevant set of image files and the core.*pid* file. The **restart** system call, essentially a generalized **exec**, restarts a process, family, or process group from a set of process images. Each checkpoint may be generated either on request by an executing program, or at regular intervals defined by the program or an external agent. An executing program can suppress or intercept scheduled checkpoints. (In a garbage-collected language environment, for example, the runtime system could arrange to compact storage before checkpointing.) Several mechanisms for control of periodic checkpointing are under consideration; this is a topic of ongoing work.

Unlike terminal checkpoints, the checkpointing of continually running process aggregates (families or process groups) requires synchronization to ensure consistency. In order to obtain a consistent snapshot of an executing multi-process program with shared memory, it is necessary to suspend execution of all processes of the job, copy the image files, and then resume the processes.

We would like to provide a checkpoint/restart facility which is useful in a general setting including interactive parallel applications and utilities such as editors (for recovery),

debuggers, and conceivably login sessions (shells), as well as long-running computational "batch" jobs. Because disk I/O will be the critical expense, and because process memory will often remain largely unchanged between checkpoints, we plan to investigate the feasibility of implementing copy-on-write in the file system to reduce redundant disk writes and speed up checkpoints (and other file operations).

4. Performance and Implementation Issues

The memory management model presented raises a number of implementation issues, several of which are considered in this section. Crucial efficiency questions introduced by the use of image files are addressed, after which aspects of the implementation of logical and physical segments are discussed very briefly.

4.1. Performance of memory allocation with image files

Several clear performance benefits derive from the above-described approach to memory management. Because file I/O buffers and process memory regions compete for physical memory resources, the system can adapt to dynamic requirements, in particular to maintain effective file cacheing. Memory-mapped I/O can in many cases speed up file read and write operations, by eliminating buffer copying while retaining the buffer cacheing function.

However, we have described a system in which every request for allocation of process memory must be preceded by a file creation. This would seem to impose unacceptable overhead. Furthermore, image files are generally very transient, and it is undesirable to clutter the file system with many of these temporary files. In order to see how these problems can be avoided, we must first consider exactly what factors contribute to the expense of file creation and deletion operations.[11] The major suboperations are:

- Path searching to find the inode for, and open if necessary, the parent directory for the file to be created.
- Reading the entire directory to check whether the file already exists.
- Allocation and initialization of an in-core inode.
- Allocation of disk inode and writing inode to disk.
- Updating directory with new file name.
- For delete: freeing of any allocated data and indirect blocks, removal of disk inode, removal of file name from directory.

[11]In some swapping implementations, the lack of contiguity of data blocks in the file system further increases the expense of swaps. This can be addressed using a file system which supports contiguous pre-allocation of files or contiguous extents. In the current section we assume an architecture in which page frames are normally discontiguous in memory.

The use of reference directories for image files eliminates the path search overhead. Further optimizations stem from two observations concerning image files:

(1) There is no need for a physical segment (or image file) to have a name if there is no reference to that name. This will generally be true of private or inherited segments, and is often true of dynamically-shared segments because sharing processes can in many cases **mapin** an image file by importing an open file descriptor.[12]

(2) If the physical segment is never flushed to disk, because no pageouts, swapouts, or checkpoints occur before the segment/image file is deleted, then there is no need for the existence of the image file to be registered on the disk at all.

These are exploited using a strategy of *lazy file creation*. When a file is newly created in this mode, no inode need be written to disk until the first actual I/O operation, and no name need be created in the directory until it is needed for a subsequent open or until the directory is itself read (e.g., via ls).[13] In fact, no file name need even exist until it is actually used.

Lazy file creation is requested with a new flag option in **open**, which causes files to be created *anonymously* in a specified directory. The meaning of the anonymous option is that a new filename is generated by the system; the path argument to **open** specifies a directory only. The following call creates an anonymous file in the reference directory *@image*, though any directory name could have been used.

open("@image", O_CREAT | O_ANON)

The file descriptor of the anonymous file is returned. For most references to the open file (e.g., **mapin**), this is sufficient. A file name is generated and installed by the system only when the *@image* directory is read or explicitly requested by the call,

fdname(*fd, fname, size*).

Creation of the incore inode remains as the only operation in the above list which must *always* be performed when a file is created for **mapin**. In Symunix II, a mapped-in inode is, in fact, the primary locus of reference and control for a physical memory segment. The internal organization of the memory management component is described briefly in Section 4.4. As of this writing, no data exists for evaluating the overall performance ramifications of this system.

---

[12]This is made possible through a new system call, pdup, which *dup*'s open file descriptors across processes running under the same userid (see [ELS88]).

[13]Since it is possible that a name is created for a file that has no inode on disk, the fsck utility must be able to properly handle certain cases of dangling pointers.

4.2. TLB coherence and demand paging

One approach for solving the TLB consistency problem on multiprocessors (see [TKS88], [RTY87]) is "TLB shootdown", in which a processor making certain non-local mapping changes must broadcast a change to other affected processors and wait for them to update their TLBs and respond before proceeding. Because of the potential expense of this mechanism, especially on large multiprocessors, Symunix II avoids non-local mapping changes entirely. Each operation specified in the above interface affects only the current process. Global mapping changes are accomplished in user mode, where synchronization can usually be avoided.

This policy mandates a somewhat conservative approach to paging. Pages can never be evicted while mapped into multiple active processes. The current implementation is a hybrid of swapping and paging, in which no process can run until sufficient memory is available for the entire process image. However, page frames are not actually allocated or initialized until referenced. On swapout, pages are placed on an available list from which they may be reclaimed by the original process or allocated (and written out to the proper image file if modified) by another process.

4.3. Copy-on-write

Copy-on-write is not truly a form of memory sharing, but rather an optimization useful on some architectures when logically private copies of memory areas are created. It is invoked in Symunix II in three circumstances:

(1) On **fork** or **spawn**, when a logical segment is marked to be copied to child processes.

(2) On **exec**. By creating text and initialized data segments as copy-on-write copies of the *a.out* file, we retain the advantages of shared text when the text is read-only, and resolve two traditional drawbacks to shared text. Because any text page can be privately modified without affecting other sharers, breakpoints can be implanted by a debugger, and the disk file can be rewritten, while the file is shared. Thus **ptrace** becomes more flexible, and the notorious ETXTBSY error (can't open file for output because it is a shared text program being executed) is eliminated.

(3) On **read**. When an image file is read with the **read** system call and the relevant page is resident, or when multiple processes read the same block of a file, the copy-on-write optimization is appropriate.

Our TLB coherence policy will occasionally limit the use of copy-on-write, in particular on **read** operations. If any region involved in a copy is already shared, then it is impossible to set the pages to read-only status (so as to catch writes) without non-local TLB changes.

4.4. Logical and physical segment representation

Each process has an *address space table*, which contains a list of *logical segment descriptors* (*lsegs*) for the process. Each lseg stores the virtual address and size of a logical segment, virtual attribute information, a pointer to the inode of the image file representing the physical memory segment, and the offset into the image file. The inode contains a count of referencing lsegs. Buffer headers for memory-resident pages of the image file are accessible directly from the inode, indexed by file offset. The buffer headers perform their conventional role with respect to I/O, and at the same time represent pages of the corresponding physical segment. Each buffer header contains a pointer to a page frame which holds the buffer data; in the presence of copy-on-write several buffer headers may share a single frame.

The machine dependent layer manages page tables or other structures needed for a particular architecture. Memory management structures are associated with their corresponding logical segments through a hashing mechanism. The hash function is designed to facilitate sharing of structures (e.g. page tables) among multiple lsegs which share physical memory segments whenever possible. When two processes have identical virtual-to-physical mappings with identical attributes, the mapping structures (e.g. segment and page tables) are entirely shared, as long as such sharing is permitted by the architecture.

5. Status

Symunix I has been running on an 8 processor Ultracomputer prototype since summer, 1984. A preliminary version of Symunix II, with the new memory management interface but without image files, is currently being debugged on the Ultracomputer prototype, and on a single processor RP3 simulator. (The current system is symmetric, but not highly parallel because it relies on very large critical sections.) We plan to complete the implementation of image files by early summer, 1988.

6. Acknowledgements

The ideas described in this paper have benifitted from many discussions with other members of the Ultracomputer Research Laboratory at NYU and, also with researchers at the IBM RP3 project. Eric Freudenthal contributed to the design of the high-level memory management library, and Vicki Myroni is porting the system to the IBM RT PC. We also thank Wayne Berke for careful reading of the paper.

REFERENCES

[Ba78]

Baudet, G. M., "Asynchronous Iterative Methods for Multiprocessors", *Journal of the ACM*, Vol. 25, No. 2, pp. 226-244, April 1978.

[BO87]

Beck, B., Olien, D., "A Parallel Process Model", *Winter USENIX Conference Proceedings*, 1987, pp. 83-101.

[EGL86]

Edler, J., Gottlieb, A., Lipkis, J., "Considerations of Massively Parallel UNIX Systems on the NYU Ultracomputer and IBM RP3", *Winter USENIX Conference Proceedings*, 1986.

[ELS88]

Edler, J., Lipkis, J., Schonberg, E., "Process Management for Highly Parallel UNIX Systems", 1988 (in preparation).

[GMS87]

Gingell, R., Moran, J., Shannon, W., "Virtual Memory Architecture in SunOS", *USENIX Proceedings*, June 1987, pp. 81-94.

[GLR83]

Gottlieb, A., Lubachevsky, B., Rudolph, L., "Basic Techniques for the Efficient Coordination of Very Large Numbers of Cooperating Sequential Processors", *ACM Transactions on Programming Languages and Systems*, Vol. 5, No. 2, 1983, pp. 164-189.

[Got87]

Gottlieb, A., "An Overview of the NYU Ultracompute Project", *Experimental Parallel Computing Architectures*, ed. J.J. Dongarra, North Holland, 1988, pp. 25-95.

[HK86]

Hoel, T,. Keller, B., "A Unix-based Operating System for the Cray 2", *Winter USENIX Proceedings*, 1986, pp. 219-224.

[Kil84]

Killian, T.J., "Processes as Files", *USENIX Summer Proceedings*, 1984, pp. 203-207.

[OCD88]

Ousterhout, J.K., Cherenson, A.R., Douglis, F., Nelson, M.N., Welch, B.B., "The Sprite Network Operation System", *IEEE Computer*, Feb., 1988, pp. 23-35.

[PBG85]

Pfister, G., Brantley, W., George, D., Harvey, S., Kleinfelder, W., McAuliffe, K., Melton, E., Norton, V., Weiss, J., "The IBM Research Parallel Processor Prototype: (RP3): Introduction and Architecture", *Proceedings of the 1985 International Conference on Parallel Processing*, Aug. 1985, pp. 764-771.

[RTY87]

Rashid, R., Tevanian, A., Young, M., "Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures", *Second International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 1987, pp. 31-39.

[TKS88]

Teller, P., Kenner, R., Snir, M., "TLB Consistency On Highly-Parallel Shared-Memory Multiprocessors", *Hawaii International Conference on Computer Systems*, 1988.

[TR87]

Tevanian, A., Rashid, R., Young, W., Golub, D., Thompson, M., Bolosky, W., Sanzi, R., "A UNIX Interface for Shared Memory and Memory Mapped Files Under Mach", *USENIX Proceedings*, June 1987, pp. 53-67.